NLP with CLTK

Pre-Workshop-Workshop

Sarah Lang

Nov. 2019

Contents

1 Natural Language Processing (NLP)

Terms and sub-fields

Quantitative methods in literary studies began in the sub-field of Stylometry: L.A. Sherman *Analytics of Literature* (1893). Now, there are many sub-fields and different names for quantitative methods for texts and language processing[a] in/for the Humanities:

Quantitative Text Analysis (QTA)

Computational Literary Studies (CLS)

Computational Linguistics

Text Mining

Franco Moretti's *Distant Reading*

Matthew Jocker's *Macroanalysis*

[a] A fascinating introductory read on the 'statistical language model', underlying computerized language processing, is Wu 2018, Don't be put off by the emphasis on mathematics, it's really a series of easy-to-understand blog posts about langauge processing.

Functions of QTA

1. **exploratory**
   Exploring a corpus for potentially interesting patterns. Be pointed to interesting phenomena; find new research questions; explore the 'potentials' of the text/corpus.
2. **descriptive**
   Describing phenomena quantitatively which we already 'feel' are there but we want to grasp them more 'objectively'. Confirm 'qualitative hypotheses quantitatively.
3. → both are not 'results'! They still require interpretation to be valid for Humanities purposes.

2 Intro to NLP concepts

Basic concepts I

Bag-of-words (BOW)

type* vs. *token

To be or not to be. = 6 *tokens*, 4 *types*. → aus den *types* wird der *bag of words* aufgebaut, darin werden dann alle Vorkommnisse (*tokens*) gezählt.

type
beschreibendes Kriterium

token
Analyseeinheit

case-folding
alles in *lowercase* analysieren: hat vor und Nachteile (z.B. Eigennamen), am Satzanfang evtl. sinnvoll, außer für Stilometrie. Stattdessen evtl. ***truecaseing***

Je nach Entscheidung bzgl. *case-folding/truecasing* ist ein Wort in Großschreibung ein anderer *type* als in Kleinschreibung!

Bag-of-words

Was geht dabei alles verloren?

- Wortstellung
- Zusammenhang
- Phrasen
- Reihenfolge
- Ironie, Sarkasmus, Negation

Basic concepts II

Zipf'sche Verteilung (*Zipf's Law*)
starker (proportionaler) Abfall der Häufigkeit (und absoluten Streuung) mit zunehmendem Rang

> Zipf's law states that given a large sample of words used, the frequency of any word is inversely proportional to its rank in the frequency table. (Wikipedia)

Stopwords
überproportional häufige Wörter (und, *the*, etc.). In Listen gesammelt, meist aus der Analyse entfernt, da sie die interessanten Partien 'verdecken'. *To be or not to be* = alles Stopwords.

Wortfrequenzanalysen
siehe 'Statistische Grundlagen'

Unigramme Bigramme n-Gramme

Unigramme vs. n-Gramme

Unigram: Jeglicher Kontext geht verloren.

n-Gramme: ***sparse data*-Problem**

d.h. wir haben fast nie genug Daten dafür, außer bei *sehr* großen Korpora. Die Häufigkeiten werden dabei schnell so 'selten', dass nichts statistisch Relevantes mehr dabei ist. Mehr als 3-5-Gramme sind daher nicht üblich. (zumal für historische Sprachen ohnehin viel zu kleine Korpora erhalten sind!)

Basic Concepts III

KWIC (=Keyword in Context)

Kollokation

Konkordanz

> Unter Konkordanz (zurückgehend auf lat. *concordare* „übereinstimmen"), versteht man in den Textwissenschaften traditionellerweise eine alphabetisch geordnete Liste der wichtigsten Wörter und Phrasen, die in einem schriftlichen Werk verwendet werden. Der Begriff stammt aus der Bibelwissenschaft. […]
>
> Konkordanzen sind heute in der Regel elektronisch erstellte Trefferlisten, die sich aus der Suche meist nach einem Wort oder einer Phrase, eigentlich aber aus der Suche nach jeder beliebig definierbaren Zeichenkette ergeben. In einer Konkordanz ist meistens auch die nächste sprachliche Umgebung des gesuchten Ausdrucks, der sogenannte *Kontext* angeführt, also beispielsweise der gesamte Satz, in dem ein gesuchtes Wort auftritt.
>
> Als Synonyme für Konkordanz gelten fallweise die Ausdrücke *Register* und *Index* oder *Index verborum* („Verzeichnis der Wörter").
>
> In der Korpus- und in der Computerlinguistik haben sich zudem, auch im Deutschen, der Ausdruck *Key Word in Context* sowie dessen Abkürzung *KWIC* als Benennungen für den in einer Konkordanz angezeigten Suchbegriff eingebürgert. (Wikipedia)

Sentiment Analysis

Sentiment Analysis

Eine weitere bekannt gewordene 'Wörterbuch-Methode'. Man macht quasi ganz normal die Auszählung und verbindet dann die Tabelle per *join* mit einem *sentiment dictionary*, das Wörter in Kategorien bewertet. Allerdings passieren natürlich Missverständnisse: Bsp: Jane Austens häufigstes Nicht-Stopwort ist 'Miss', aber in *lowercase* (da *case-folding*) wird es natürlich fehltinterpretiert als 'vermissen' und damit als negatives Gefühl gewertet!

3 Statistische Grundlagen und Theorien

Statistische Grundlagen

siehe Kap. 20 DH-Einführungsbuch: Quantitative Analyse[a]

Merkmalserhebung

z.B. Worthäufigkeiten, Anzahl Orts- und Personennamen, etc. → in Tabelle festgehalten, wo jede Spalte ein Merkmal darstellt. →

Merkmals-Matrix

Strukturierung, in Tabellenform für Computeranalysen leicht zugänglich, aber auch Informationsverlust durch Zerstückelung.

Absolute und relative Häufigkeit / Frequenzanalyse

Absolute Häufigkeiten bringen nichts im Vergleich zu anderen Werken, die ja nicht gleich lang sind. → Feststellung der relativen Häufigkeit eines Wortes per 1000 Wörter.

Median / Mittelwert

Damit man werkintern mit der relativen Häufigkeit etwas anfangen kann, könnte man sie mit dem Median dieses *types* im Gesamtkorpus vergleichen. Diese nennen sich dann *Maße der zentralen Tendenz*, weil sie die Verteilung mehrerer Werte in einem Wert zusammenfassen.

Streuungsmaße

Nur weil ein Werk so und so relativ häufig ist, sagt das ja nichts darüber aus, ob das Wort einfach ganz oft an einer Stelle vorkommt oder durchgehend viel verwendet wird. → Schwankungen messen.

Standardabweichung

Pro *type* beschreibt sie die Streuung: "Wurzel aus Mittelwert der quadrierten Abweichungen jedes Einzelwertes vom Mittelwert aller Werte".

Mittelwert-Normalisierung

Zipf's Law: starker Abfall der Häufigkeit (und absoluten Streuung) mit zunehmendem Rang → Vergleichbarkeit zwischen Wörtern erschwert. → Abziehen des *type*-Mittelwerts (der Sammlung) von jeder Häufigkeit im Einzeltext.

z-scores transformieren

um Streuung vergleichbar zu machen: Mittelwert ist dann immer 0, Standardabweichung 1. → Hinweis auf das Verhalten des *types* in der Sammlung → Merkmalsskalierung.

Statistische Signifikanz

Statistisch *signifikant* ist schnell mal etwas. Nicht jedes signifikante Outcome ist auch *relevant*. Und sagt das überhaupt etwas aus? Nein, ohne Interpretation nicht!

> Wann haben wir ein relevantes Resultat? Wir müssen vorher definieren, was «Relevanz» bedeutet. Was ein relevanter Unterschied ist, hängt ab vom Fachgebiet/Fachwissen. Die Statistik hat hier keine Antwort! (Quelle)

[a] vgl. Schöch 2017b.

Konzepte bzw. Theorien der QTA

> Reading 'more' seems hardly to be the solution. Especially because we've just started rediscovering what Margaret Cohen calls the 'great unread'. [Apart from "its canonical fraction, which is not even 1 per cent of published literature"] there are [thousands of books] – no one really knows, no one has read them, no one ever will.[a]

Franco Morettis *Distant Reading*

Vorstellung serielles Lesen vs. menschliches *Close Reading*. Behauptet, Literaturwissenschaft wäre nicht mehr vollständig ohne quantitative Aspekte. Das serielle Lesen aber auch irgendwie 'als Alternative' anstatt *close reading*.

> […] you *reduce* a text to a few elements, and *abstract* them from the narrative flow, and construct a new, *artificial* object. […] And with a little luck, these maps will […] possess 'emerging' qualities, which were not visible at the lower level. […] Not that the map itself is an explanation, of course. It offers a model […][b]

Matthew Jockers's *Macroanalysis*

Vorstellung einer Zoom-Bewegung: Die quantitative Analyse bietet Anstöße für das *Close Reading*, etc. Er fordert die Unterscheidung zwischen *reading* und *analysis* – der Überblick 'aus der Ferne' ist für ihn nicht 'Lesen', wie etwa Morettis Benennung suggerieren würde.

1. Kontextualisierung durch das *zooming out*
2. andererseits aber auch extremes *close reading*: So viel Details wie dem Computer können einem Menschen fast gar nicht auffallen, weil wir Details ja gar nicht so richtig wahrnehmen.
3. besser informiertes Verstehen der Primärtexte: die *macroscale* gibt weniger 'anekdotische' Beweise als das sehr genaue Lesen nur eines einzigen Texts.
4. 'harvesting findings, not facts'

'Mixed Methods'-Ansatz, Wechselspiel

> This is not close reading; this is macroanalysis, and the strength of the approach is that it allows for both zooming in and zooming out.[c]

Pipers *cultural analytics*

[a] Moretti 2013, 45.
[b] Moretti 2005, 53.
[c] Jockers 2013, 23.

4 NLP-Pipeline und Korpusaufbau

NLP-Pipeline

(*Pre-Processing*)

→ fällt in unserem Fall normalerweise unter das sog. (*Pre-Processing*)

Tokenizer

Einfachste Tokenizer: Nur der Whitespace/Leerzeichen ist Trenner. Erweiterbar durch Hinzufügen von Punktuation oder sprachinternen Spezifika.
Sprachgebunden, z.B. frz. *l'enfant*, lat. *dixitque*, en. *don't* auflösen, …
Wie wird mit ***compounds*** (zusammengesetzen Wörtern) umgegangen? Fremdsprachliche **Lehnwörter**, die zusätzlich noch stehende Wendungen sind (z.B. *en masse*)?
Geht meist ohne Wörterbuch – für die meisten anderen NLP-Methoden müssen Wörterbücher und Grammatiken vorliegen (! ist für einen historischen Sprachstatus absolut nicht selbstverständlich) → Computer kann nicht denken und ist z.B. mit der uneinheitlichen Orthographie von frühneuzeitlichem Deutsch völlig überfordert: Methoden davon teilweise in die völlige Impraktikabilität reduziert!
→ aber geht natürlich super für moderne lebendige Sprachen, v.a. Englisch.

> **But what is a word?** We tend to think of a word as a unit of meaning that often has an analog in the real world. […] A computer doesn't know what a word is and certainly has no sense of what words might refer to. […] Words are usually bounded by spaces and punctuation, and a computer can be told to split a long string (text) into shorter strings (words) by looking for the demarcation characters […] – a process called tokenization.[a]

[a] Sinclair and Rockwell 2016, 283.

Im Detail dazu im Praxisteil

Lemmatization

Mithilfe von **Wörterbuch/Vokabular, Grammatik und morphologischer Analyse** der Wörter die Grundform (das *Lemma*, wie man es im Wörterbuch hat) zu finden, damit alle Wortformen korrekt als Formen desselben *type* gezählt werden können.

Stemming

Wortende bis zur Wurzel abschneiden, nicht unbedingt auf linguistisch korrekte Art und Weise.

Parsing

POS tagging

***Named Entity Recognition* (NER)**

Korpusaufbau I

DH-Einführung: Kap. 16 'Datensammlungen'[a]
Datensammlung nicht mit der technischen Umsetzung (Datenbank, etc.) verwechseln! ('Base of data' ≠ 'database')
Ablauf: Abgrenzung des Gegenstands, Auswahl der Datensätze, Zusammenführung, Vereinheitlichung, Erheben von Metadaten (Informationen, die die Datensätze beschreiben), (ggf.) Verfügbarmachen.
Definition des Gegenstands der Datensammlung in Abhängigkeit von einer Forschungsfrage. **Mögliche Kriterien:** zeitlich, räumlich, Epoche, Genre, AutorIn / Urheber, Umfang, technische Verfügbarkeit.

Grundgesamtheit

Gegenstandsdefinition → Grundgesamtheit, also Gesamtmenge aller relevanten Gegenstände kann festgestellt werden. Wie steht mein ausgewähltes Korpus zu dieser Grundgesamtheit *aller* in Frage kommenden Objekte? Ist es genug, um aussagekräftige (statistisch signifikante) Schlüsse zu ziehen? Im Zweifelsfall kann man immer die Fragestellung eingrenzen.
Vollständige (!) Datensammlung: Wenn ich mich mal entschieden habe, muss ich auch konsequent dabei bleiben, sonst wird es nichts!
Frage nach Verfügbarkeit: Aber falls verfügbar Frage nach Qualität der verfügbaren Inhalte (!) bzw. Nutzungsrechte, etc.

[a] Schöch 2017a.

Korpusaufbau II: Auswahlprinzipien

Repräsentatives Sample

Umfang kann sich z.B. durch eine repräsentative Stichprobe (*sample*) begrenzen:

> "Representativeness refers to the extent to which a sample includes the full range of variability in a population." (Biber 1993, 243)

→ auf Grundlage der Stichprobe gültige Aussagen über die Grundgesamtheit treffen.
Gleichbehandlung aller Werke oder Vorzug dem Kanon / weiter verbreiteten Werken? Auch die 'Ausreißer' mit aufnehmen? Per Zufall oder 'Gütekriterien'?

Balancierte Sammlung

Alternative zum repräsentativen Sample. Gezielte Konstruktion nach wesentlichen Kriterien (abhängig von der Forschungsfrage). Korrelationen, also statistische Abhängigkeiten, zwischen Eigenschaften der Datensätze und den ausgewählten wesentlichen Kriterien sind zu vermeiden.

Opportunistische Wahl

Was ist verfügbar?

> A *corpus* is a body of texts (though a corpus can have only a single text). […] One size does not fit all. […] A digital corpus is a bit like a bag of Lego where pieces can be built up in various configurations.[a]

[a] Sinclair and Rockwell 2016, 281–282.

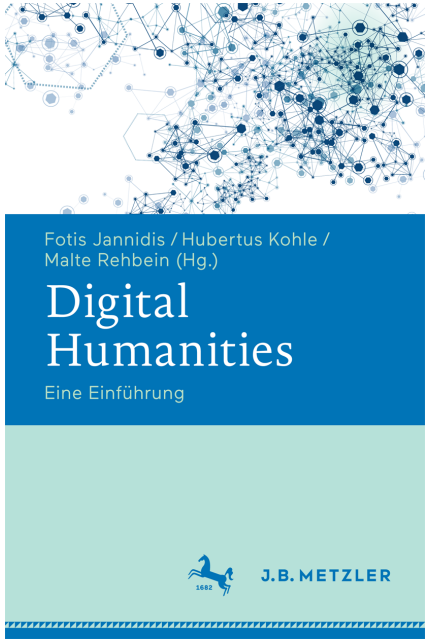
Korpusaufbau III: Repräsentativität

Repräsentativität

Was wird digitalisiert und warum? Zufall? Hat 'der Kanon' Priorität? Sind Texte verlässlich und verfügbar oder aber aus rechtlichen Gründen unzugänglich? Wir können leider nicht *restlos alles* digitalisieren. Sind dann digitale Massenanalysen überhaupt repräsentativ? Vgl. GoogleBooks.

> By allowing a researcher to survey a larger set of documents, text mining promises to give a picture […] that is "more representative" than an account based on a few hand-selected examples. But representative of what? Digital libraries […] do not include every book ever published. Moreover, even if we had a copy of every book, the print record itself would not reflect the demographic reality of the past, since access to print has been shaped by class, gender, and race.[a]

[a] Jockers and Underwood 2016, 300–301.



5 Tools und Visualisierung

Informationsvisualisierung (*'DataViz' / 'InfoViz'*)

Siehe Kapitel 23 'Informationsvisualisierung' (DH-Einführungsbuch)[a]
Grafische Elemente als Kommunikationsmittel ● visuelle statistische Darstellungen als Repräsentationen von Sprache ● Entlastung des Geistes bei großes Datenmengen: *snapshot* statt Gesamtwerk. ● 'method for seeing the unseen' ● durch die Visualisierung werden abstrakte Daten räumlich angeordnet ● oft interaktive Schnittstellen, um den Denkprozess zu begleiten

Funktionen

1. **anschauliche Präsentation** von Daten/Ergebnissen, am Ende des Forschungsprozesses
2. **konfirmative Analyse:** Sichweisen (*views*) auf Daten erzeugen, die erlauben, Hypothesen zu verifizieren oder falsifizieren. Teil des Forschungsprozesses.
3. **explorative Analyse:** nicht hypothesengetrieben, Strukturen oder Trends erkennen. → interaktive Werkzeuge, zu Beginn des Forschungsprozesses
4. **Visualisierung** (z.B. Storytelling)

makros vs. mikro

Einerseits Aufzeigen übergeordneter Strukturen ('distant reading'), aber auch Detaileinblick in die Daten.

Vorgehen

raw data → (Vorverarbeitung) → *data tables* → *visual structures* → *views*.
→ Tufte 1983: *A silly theory means a silly graphics.*

[a] vgl. Rehbein 2017.

Nie vergessen

> *A fool with a tool is still a fool!* (Grady Booch)

DataViz II: Links zu Tools

Intro to WordClouds in R ● Detailerklärung zur WordCloud selbst, aus derselben Quelle ● wordcloud-Package-Dokumentation WordSeer (Berkeley (Visualisierungs- und Textanalysetool, muss installiert werden) ● Textarc-Visualisierung (evtl. nicht verfügbar) ● Lexos: Von Pre-Processing über Analysen (mehrere Texte) bis zur Visualisierung (z.B. Rolling Window). ● Text Visualization Browser ● Overview-Tool (Download) ● Tapor-Toolübersicht ● Deutsches Textarchiv Hilfe

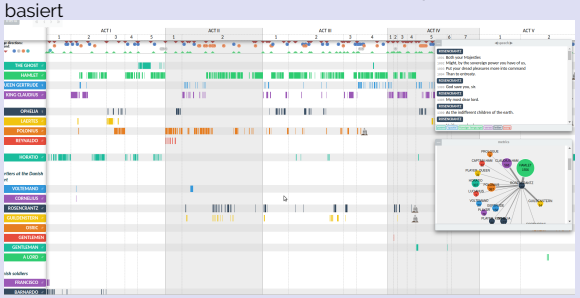
DataViz III

Frage nach der Aussagekraft der Daten

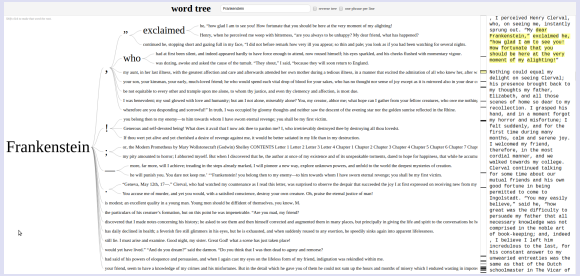
> Nur aus der bloßen Existenz von Daten kann nicht auf deren Bedeutsamkeit geschlossen werden, und nicht alles, was (in einem bestimmten Kontext) bedeutsam ist, hinterlässt messbare Datenspuren.
>
> Ein weiterer Schritt ist durch den Forscher selbst zu vollziehen: Visualisierungen sind häufig gut geeignet, um Strukturen und Muster zu zeigen. Sie liefern allein für sich aber keine Erklärungen. Mit anderen Worten: Korrelationen oder Koinzidenzen, wie sie sich vielleicht in den Daten erkennen lassen, bedeuten noch keine Kausalität, eine Wiederkehr noch keine Gesetzmäßigkeit. Erklärungen lassen sich erst durch Interpretation der Visualisierungen, oft unter Bezugnahme von anderen Daten, Quellen oder auch Methoden, durch den Forscher folgern. *(Rehbein 2017, 341–341)*

Rolling Window

To See Or Not to See: Shakespeare-Visualisierung ● Annotations-basiert

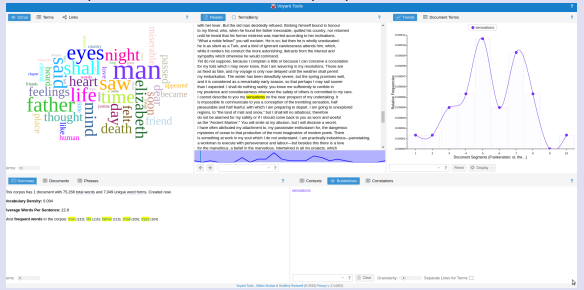


WordTree



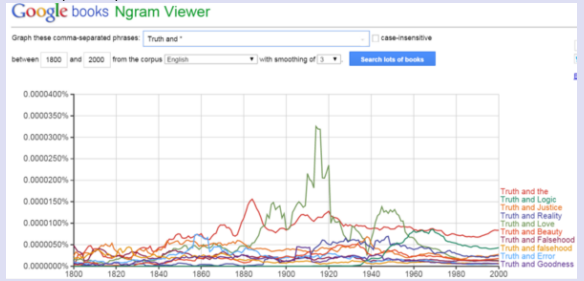
Voyant Tools

Blogpost zu Voyant ● features many standard functions for single and multiple documents ● XML input possible as well



Google Books N-Gram Viewer

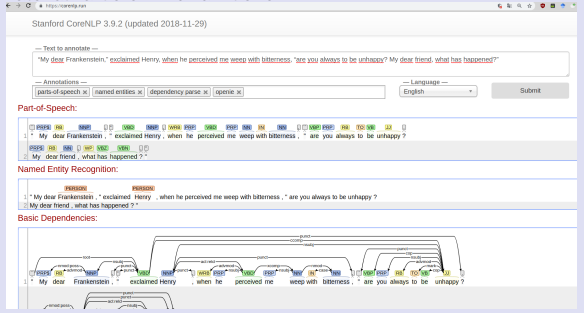
Shai Opir: Bsp. zu 'Truth'



NLP-Tools

Stanford CoreNLP

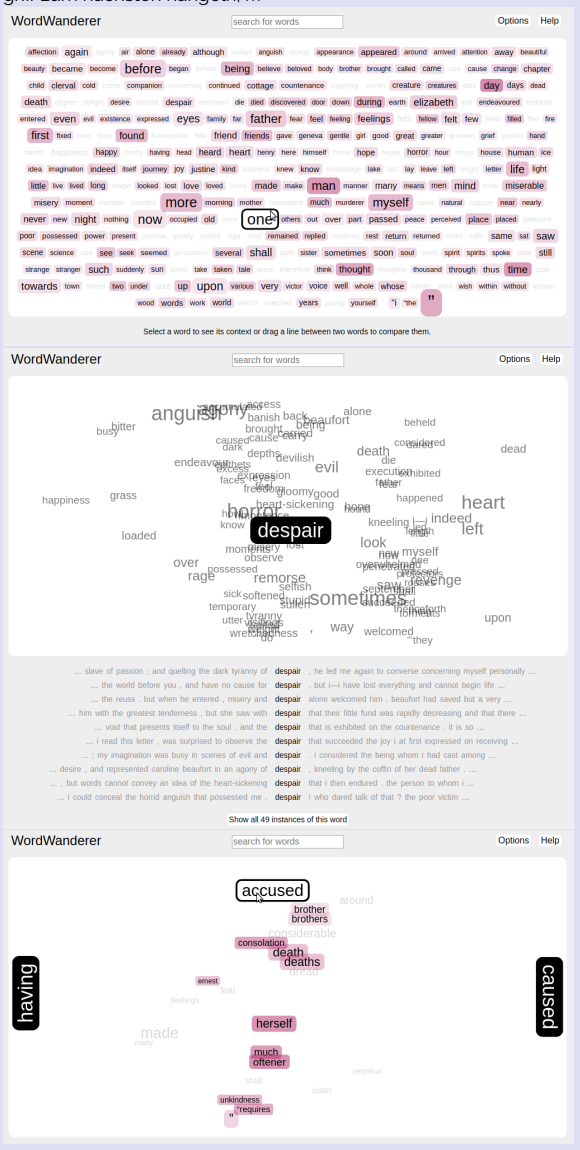
Stanford CoreNLP ● Online-Tool für CoreNLP ● Einsteiger-Tutorial zu CoreNLPs Funktionen



Word and Phrase: Konkordanzen, Frequenzlisten, Texte analysieren (ein ganzer Roman scheint ihm zu viel zu sein)
Weblicht

WordWanderer

Konkordanz, Beziehungen zu anderen Wörtern, etc. Vom einen Begriff zum nächsten hangeln, …



6 Grundbegriffe der Programmierung

Communicating with computers I

Algorithm

Recipe providing a formalized way to automate carrying out a task. Doesn't have to be noted down in a particular programming language yet, can be in 'pseudo-code' ("Whenever you find a verb, do this. When you find a noun, do that." etc.)

Algorithmic thinking

The existence of this term hints at the fact that computers don't think like humans do, in fact, they don't think at all. A beginner needs to get to know ways of communicating effectively with a computer. This involves experience, but at a higher level also requires a certain background knowledge about the inner workings of computers (i.e. what are data types, how are libraries built, what parameters and arguments do functions expect and what do they return?)

Communicating with computers II

Source code and compiled code

Source code is what we see and write. However, a computer can only understand machine language (1s and 0s), so we need to 'bridge this gap'. There are two ways of doing this: Either, we need to 'press a button' (which might not always be a litteral button in a normal programming language, but is – for example – when you compile LaTeX code into a PDF on Overleaf…) and then the code gets compiled. It produces an executable as output which can, in turn, be executed. Instead of compiled and interpreted, we can also say static and dynamic. Or we use an 'interpreted' language where this 'compiling process' is done, line by line, while to code runs. Python is such an interpreted language. 'Interpreting' at runtime is a bit slower and it can mean that the compiler won't be there at (the non-existant) compile time to inform you about some inconsistencies in your code. But in this case, you will have to 'run' it more frequently anyway, so it probably doesn't make a difference and standard code editors will help with pointing out inconsistencies anyway. You basically get your error messages one way or the other. Which leads us to the next point…

Debugging errors and error handling I

Debugging

is the act of cleaning up your code when there are errors in it which prevent it from working correctly, as intended or from running at all. The term comes from a time when computers were huge an litterally had to be debugged because bugs were stopping the wires from working correctly. There are different types of errors, most importantly:

Syntax errors

mean that you didn't respect the conventions of your programming language. Like spelling and grammar mistakes in normal writing.

Logic errors

are hard to detect since the conventions of the programming language are respected and automated checking tools might not find them. An example would be passing nothing to a function (indirectly and thus, invisibly) and, thus (obviously but not obviously) getting incorrect results ('Null reference error', usage of wrong conditional operator = design flaw). Usually these are 'bigger' errors in the program flow compared to the more localized syntax errors (typos, missing semicolon or closing brackets, etc.).

Debugging errors n error handling II

Semantic errors

mean that the code runs alright, but it doesn't do what you intended it to do. Remember that the computer will do exactly what you tell it to do even if this is not, in fact, the same as what you intended to do. (This happens a lot, not only to beginners).

Runtime error

A program crashes due to unexpected input but the compiler can't know someone will input something which will cause the crash (such as an unexpected and thus uncaught division-by-zero error which you should actually prevent). There are many recurring types of errors which consequently have received their own names, such as index-out-of-range error or off-by-one error, to name to most well-known examples.

Debugging errors n error handling III

Error handling

Errors should be handeled in your program flow, this means they should be anticipated, 'caught' or 'thrown'. An error message will specify what kind of problem happened which makes debugging easier. An expected and thus properly handeled error prevents the program from crashing whenever something problematic happens (which shouldn't happen!). If you have an advanced error handling in place, your program will know how to react when different types of errors happen. However, this is maybe overkill for tiny script-like programs where a crash is not an issue.

IO

Input and Output

Input and output (sometimes referred to as I/O) are base concepts of how computing works: You give input to a function or computer program and expect it to return a certain output. In between, some processing happens.
This is actually the same principle in a function you know from maths, if this helps you. For example, in image processing, every pixel of an image has a colour code stored in some data format (which you don't need to worry about). Just know that it has a numeric value which can be used as input for a function. So when processing an image in an image manipulation programme (such as turning it to grayscale), every colour value gets inputted into a function which makes some decisions, such as: If the colour value is above threshold XY, make it black - if it is below it, make it white. So in the "output graph", like a function graph in maths, you get these transformed values which can, in turn, be "put back" (written onto) their old locations in the coordinate system that is the grid behind what you see as a digital image.
The following terms will give you more detailed understanding of how exactly these functions work, but keep the image of maths functions in the back of your mind. Computer functions are not exactly the same, but the principle is similar.

Abstraction

Parameter

is a variable in the declaration of a function. Together with variables (and parameters *are* a type of variable), parameters contribute to the abstraction of functions. This is a good thing.
Imagine you would (and you will in the process of developping a function) write multiple lines of code. Once you got the thing working, you want to apply it to all the other things this automated processing needs to be applied to. If you don't create a function where this gets abstracted, you have to copy all those lines over and over again, only to make minor changes in the actual code. This is a situation that practically screams "Define a function for this repetitive behaviour!" at you. You should listen to the call. Once it's abstracted and you realize a change needs to be made, you need to make that change exactly once and not in every single line of the code you copy-and-pasted around which now clutters your editor. This means, with writing a function, we have reduced redundancy – and in computer things, we always want to reduce redundancy. Amen.
However, the variables and parameters are abstract stand-ins. Their names can act like placeholders. This can be quite confusing to a beginner. The point is that this abstract placeholder (parameter) in a function is not the same thing as the actual content you pass to the function (which is called the 'argument', see below):

Reuse is Reduction of reundancy

Library

A library (or package) is essentially nothing else than a bunch of functions, only that when you load them into your program, you can profit from (usually) really well-done and maintained functions somebody else made. Using libraries, you can accomplish quite something writing very little code yourself. If not for practice purposes, it is always better to reuse a standard library rather than writing things yourself – package maintainers are experienced experts and might have taken things into account which you didn't. When you use code from a library, it means that this code is not part of the standard repertoire ('vocabulary') of your language: It is a set of functions made using this standard repertoire just as your own functions are. When you invoke or 'load' them, it's essentially as though the computer would just copy their text into your document behind the scenes (and this is exactly what happens behind the scenes). By combining and reusing libraries, you can be really efficient and "stand on the shoulders of giants", as they say. Functions coming from libraries can have the name of their library prefixed as a 'namespace'. This can sometimes be a little confusing, especially for beginners because it obscures where which names belong to in longer expressions and requires you to know/remember from which libraries your used functions come.
Redundancy is always an unnecessary source of error. Except for in data archiving. Here, redundancy is security.

Program Flow and Flow Control I

In the beginning, we learned that an algorithm (and thus, also it's concrete *implementation*, a program) is essentially a cooking recipe which allows us to automate things which are simple enough so they lend themselves to being automated. Sometimes, even things which look complicated at first can be reduced to simpler substeps. This is called the **'divide et impera' principle** in computer programming.

Abstraction

Variable

A variable is a container for storing information. A variable is a container for storing information, kind of like in maths where a variable is a placeholder for a yet unkown value. It is named (such as 'x' in maths). Calling the name usually causes it to give you its 'contents'.

Function

A function is a generalized bit of code, made for general-purpose reuse. So essentially, you abstract what is being done. Instead of "process my document test.txt" you would write something like `process(text)` and would be able to pass a document of any name to this `process()` function like `process("test.txt")` or `process("unicorn.txt")`. This has the awesome advantage that it doesn't matter whether your document is called 'test' or 'unicorn' or, in fact, any other crazy name you can come up with. A function usually has a 'signature' that means a list of what parameters you pass to it (in the definition of a function, you say which type of data will be inputted) and what it will return.

Abstraction

Argument

is almost the same as a parameter, but the difference is that it means the actual value which gets passed to the function. So when using a function, you can pass the number 3 to it. In the definition/declaration, you say that an integer variable will be passed to it (=the parameter is an integer type value).

Return value

What a function will return as a result. This has a 'return type', specifying which data type will be returned. This can sometimes be good to know so that we don't get confused in further processing steps. However, the point about the whole function-abstraction-thing is that we really shouldn't need to know exactly how the processing is done in a function. We only need to know what we put int (input) and what we expect to get out of it (output). This makes it easier to build really complex programs where you don't want to have to worry about how excatly things were implemented in the detailled single processing steps.

Program Flow and Flow Control II

However, cooking recipes always expect the same input (the ingredients, so to speak). If the input differs from what's specified (even just a tiny little bit!), the computer really doesn't know what to do with it because the computer doesn't think and doesn't have common sense. These situations where a computer doesn't 'get' simple things and doesn't compensate for our little inconsistencies (such as typos) can be irritating. So please remember always that the computer doesn't think. It's not its fault. *You* made the mistake, so go and correct it. To make our programs more robust, we can use flow control, that is: some simple rules in the form of if-then conditions or behaviour which gets repeated as many times as needed or as long as there is still input. These regulate what we call 'program flow'. There are a few options available, see examples below.

Program Flow and Flow Control III

Loop

means repeating an action for a specified number of times. If something goes wrong in the definition of the ending condition (which always need to be defined!), you can end up with an 'infinite loop' which will cause the program to hang or 'freeze' and eventually crash. For-loops repeat for a fixed amout of times, while loops repeat while a condition is true, do-while-loops execute at least one time and then behave as while-loops.

Conditional

is an if-then-type decision which can be used to regulate program flow. You can define multiple if cases, but also if-else and else cases which will handle everything which doesn't match the condition.

Data types

Suffice it to say, for now, that there are different ways of storing data. As a beginner, especially if you're using a loosely typed language such as Python, this isn't all that important at the beginning so I don't want to be too verbose as not to confuse you with irrelevant facts.

Data types

There are simple ('primitive') data types, such as integer, characters, floating point numbers or strings; as well as complex data types such as hashmaps, lists, dictionaries, etc. which are specific to the implementation in a programming language. There are also ways for you to define your own complex data types (such as defining that if you want to store data on people, you need their names, ages and addresses and and address always consists of X, Y and Z). Python is a 'dynamically (or weakly) typed' language, so you don't need to perform specific operations yourself (as opposed to 'strongly typed' languages like C) such as:

Type-casting

means to 'change' data from one type to another. A classic example is that '1' if typed into a terminal, actually is a string, not a number to the computer. In order to calculate with it, you need to change it to an integer. But this is not an issue in Python, so don't worry about that now.

Various

Script / Scripting

Scripting in a terminal is not the same thing as programming or writing software (which contains memory allocation, error handling, etc.).

> **To get a computer to do anything, you (or someone else) have to tell it exactly – in excruciating detail – what to do. Such a description of "what to do" is called a *program***, and *programming* is the activity of writing and testing such programs. […] The difference between such (everyday) descriptions and programs is one of degree of precision: **humans tend to compensate for poor instructions by using common sense, but computers don't.** […] when you look at those simple (everyday) instructions, you'll find the grammar sloppy and the **instructions incomplete.** A human easily compensates. […] In contrast, computers are *really* dumb.[a]
>
> We also […] to assume "common sense". **Unfortunately, common sense isn't all that common among humans and is totally absent in computers** (though some really well-designed programs can imitate it in specific, well-understood cases).[b]

[a] Stroustrup 2014, 44. Hervorhebung hinzugefügt.
[b] Stroustrup 2014, 35. Emphasis added.

Programming in the DH

- mostly rather 'scripts' than actual programs (that's called research software engineering #RSE)
- you really only need enough programming knowledge to be able to:
  - read and understand other people's code
  - modify it a bit
  - basics of *debugging*
- today, programming is mostly knowing libraries and making the most of code reuse. In the early days, everybody used to write everything themselves which resulted in overall worse quality code and much time lost programming stuff somebody else had already done (and mabye worse than they had done it).
- much can be done using packages and lightweight *scripting* without really knowing how to program
- → passive knowledge more than active mastery

7 Python

Python

Hitchhiker's Guide ● Visual Intro with Turtle ● Learn Python Tutorial

> Python is an interpreted high-level programming language for general-purpose programming. Created by Guido van Rossum and first released in 1991, Python has a design philosophy that emphasizes code readability, notably using significant whitespace. […]
>
> Python features a dynamic type system and automatic memory management. It supports multiple programming paradigms, including object-oriented, imperative, functional and procedural, and has a large and comprehensive standard library. (Wikipedia)

The Zen of Python

> Beautiful is better than ugly.
> **Explicit** is better than implicit.
> Simple is better than complex.
> Complex is better than complicated.
> Flat is better than nested.
> Sparse is better than dense.
> **Readability counts.**
> Special cases aren't special enough to break the rules.
> Although practicality beats purity.
> **Errors should never pass silently.**
> Unless explicitly silenced.
> **In the face of ambiguity, refuse the temptation to guess.**
> There should be one – and preferably only one – obvious way to do it.
> Although that way may not be obvious at first unless you're Dutch.
> Now is better than never.
> Although never is often better than *right* now.
> If the implementation is hard to explain, it's a bad idea.
> If the implementation is easy to explain, it may be a good idea.
> Namespaces are one honking great idea – let's do more of those!
> (Wiki & Zen of Python & Hitchhiker's Guide to Python)

Python data types

Interactive tutorials: Hour of Python

Lists: related things

Python offers a tool called lists to keep track of related 'things', or values.

Lists

```
grades = [97, 62, 85, 76, 99, 93]
grades[0] # get by index. output: 97 (IndexError if index non-existant)
names = ['Anna', 'Bob']
container = [grades, names] # nested lists using variables
an_empty_list = []
grades.append(42)
grades.insert(2, 'spam') # [97,62,'spam',85,76,99,93,42]
# extending with + or
grades.extend(['foo'])
del grades[0:2]
grades.pop(3)
grades.remove(85) # will remove first occurrence of value (ValueError if non-existant)
# ['spam', 76, 93, 42, 'foo']
```

Dictionaries: key–value built-in to Python, for storage, to translate **keys** to **values**.

Dictionaries

```
foods = {'a': 'apple', 'b': 'banana', 'c': 'cookie'}
foods['d'] = 'dates'
y = {'spam': 'eggs'}
x.update(y) # merge y into x
foods['d'] # dates
# for the getter: KeyError if search by value
del foods['d']
```

Getting Started

If working with a terminal/prompt, '>>>' indicates the terminal is ready for you and waiting for input. In Jupyter Notebooks, get yourself the first empty cell and get going. `CTRL+ENTER` to compile. `[*]` means it's still compiling. When it's done, `*` will be replaced by a number.

Start by using Python as a simple calculator

Enter things like: `2 + 5 * 4 - 8 / 9`. See what happens, understand how braces work. If you disrespect some of Python's conventions, don't worry, you will be notified in an error message ☺

Indentation

Many languages use brackets to indicate blocks of code which belong together. Python, in its (somewhat extreme) simplicity, has neither brackets nor semicolons. Code blocks which belong together are marked out as such by indentation alone. Thus, you need to exercise special caution – if you get errors, it might be for this reason. Whereas in other languages, typical syntax errors would be missing closing brackets or semicolons.

7.1 Installing Python and Jupyter Notebooks

Working with Jupyter Notebooks I

> Jupyter Notebook App (formerly IPython Notebook) is an application running inside the browser. […] The Jupyter Notebook App can be executed on a local desktop requiring no internet access […] or can be installed on a remote server and accessed through the internet. (Tutorial)

1. Download the Anaconda Distribution, Python **3**, 64 bits.
2. The Jupyter Notebook App can be launched by clicking on the Jupyter Notebook icon installed by Anaconda in the start menu (Windows)

`jupyter notebook` ..... typing in a terminal (cmd on Windows)
`anaconda-navigator` ..... Starting Anaconda GUI (in Linux)

Working with Jupyter Notebooks II

> The astute reader may have noticed that the URL for the dashboard is something like `http://localhost:8888/tree`. Localhost is not a website, but indicates that the content is being served from your local machine: your own computer.
>
> Jupyter's Notebooks and dashboard are web apps, and Jupyter starts up a local Python server to serve these apps to your web browser, making it essentially platform independent.
>
> […] Each `.ipynb` file is a text file that describes the contents of your notebook in a format called JSON.
>
> […] A code cell contains code to be executed in the kernel and displays its output below.
>
> A Markdown cell contains text formatted using Markdown and displays its output in-place when it is run. The first cell in a new notebook is always a code cell. Let's test it out with a classic hello world example. Type `print('Hello World!')` into the cell and click the run button in the toolbar above or press `Ctrl + Enter`. (Tutorial)

Working with Jupyter Notebooks III

Alternative for experienced Python users:

Installing Jupyter with pip

Jupyter installation requires Python 3.3 or greater, or Python 2.7. […] As an existing Python user, you may wish to install Jupyter using Python's package manager, `pip`, instead of Anaconda.
First, ensure that you have the latest pip; older versions may have trouble with some dependencies:

```
pip3 install --upgrade pip
pip3 install jupyter
```

(Use pip if using legacy Python 2.) ● (Quelle)

Linux 'conda not found' error

Support:Conda not found error ● Support Navigator Issues

`export PATH="~/anaconda3/bin:$PATH` ..... Path fehlt
`conda -version` ..... check if working
`conda init` ..... initialize
`anaconda-navigator` ..... launch navigator

When started: Click Jupyter Notebook. Choose directory. New → Python 3.

7.2 Jupyter Shortcuts

general

`CTRL+ENTER` ..... run
`[*]` ..... still running
`[1]` ..... done, w/cell nr.

`ESC/ENTER` ... Toggle edit / cmd mode

command mode

`UP/DOWN` .. scroll up / down cells
`A` ..... insert new cell above
`B` ..... insert new cell below
`M` transform to Markdown cell
`Y` . transform cell to code cell
`DD` ..... deletes active cell
`Z` ..... undo cell deletion

in editing mode

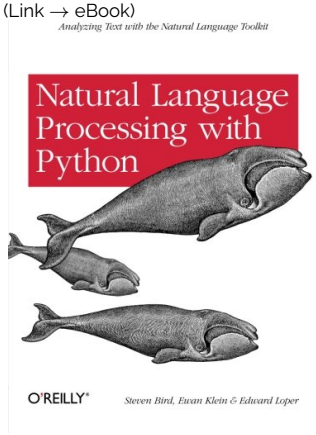
`CTRL+SHIFT+-` .. split active cell at cursor position

Markdown cells

In the Markdown cells, you can use either Markdown or HTML markup to format your notes. (Learn HTML in 5 minutes)

8 Natural Language Processing in Python

(Link → eBook)



Basics of NLTK (Natural Language Toolkit)

Natural Language versus Artificial Language (i.e. mathematical notations, etc.)
NumPy and Matplotlib have to be installed to generate plots! (should be included in Anaconda anyway)

> [Programming/Learning Python-NLTK] is just like learning idiomatic expressions in a foreign language: you're able to buy a nice pastry without first having learned the intricacies of question formation.[a]

[a] Bird, Klein, and Loper 2009, xii.

First steps in Python

```
# press CTRL+ENTER to evaluate in Jupyter
# mit '#' markiert man Kommentare, also Infos für sich selbst, die der Computer aber ignorieren soll
1 * 4 + 5 / 8
# wie funktionieren die Klammern??

# Grundlegende Funktionalitäten sind bereits in der Standard-Bibliothek enthalten. Für Spezielleres müssen wir sog. 'Pakete' hineinladen. Das geht z.B. so:
import nltk
nltk.download() # Der Schritt ist nur 1x nötig, wenn das halt noch nicht installiert ist oder out-of-date oder man nicht alles gedownloaded hat beim letzten Mal und jetzt noch andere Sachen dazunehmen will (Paket ist relativ groß, daher könnte es von Interesse sein, nur selektiv manche Sachen zu nehmen)
# nicht vergessen im Prompt/Dialog zu klicken, was man will (Progressbar startet), sonst passiert nix

from nltk.book import *
text1 # Moby Dick
```

Easy out-of-the-box things to get started

```
import nltk
nltk.download()
from nltk.book import *
text1 # Moby Dick

text1.concordance("monstrous")
text1.similar("monstrous")
text2.similar("monstrous")
text2.common_contexts(["monstrous", "very"])
# import numpy, matplotlib
text4.dispersion_plot(["citizens", "democracy", "freedom", "duties", "America"])
# NumPy and Matplotlib are needed to plot this

len(text3) # word count
sorted(set(text3)) # list unique words in sorted order
len(set(text3)) / len(text3) # number of word types / tokens = measure of lexical diversity
text3.count("smote") # word count for 'smote'
100 * text4.count('a') / len(text4) # percentage of 'a' in text

def lexical_diversity(text): # define this as function
    return len(set(text)) / len(text)

def percentage(count, total): # has to define above of the first usage!
    return 100 * count / total

lexical_diversity(text3)
percentage(4, 5)
percentage(text4.count('a'), len(text4))
```

Texts as lists of words

```
sent1 = ['Call', 'me', 'Ishmael', '.'] # Moby Dick first sentence as words in list
# stored in a variable named 'sent1'
# attention: var names are case-sensitive, so sent1 is not the same as Sent1
# some theoretically possible names (like 'not') are reserved words and thus not allowed as variable names. You will be notified by an error.
sent2 # (if you get an error which says that sent2 is not defined, you need to first type  from nltk.book import * )
sent4 + sent1 # concatenation (adding strings / text)
sent1.append("Some")
sent1 # word has been appended
text4[173] # get element via numeric index
# the first index always starts with 0, thus the last is (len(thingy) -1)
text4.index('awaken') # find out this index for a specific word
text5[16715:16735] # getting elements like this is called 'slicing'
```

Texts as lists of words

```
name = 'Monty'
name[0] # a word/string can be processed just like a list
name[:4] # get first 4 characters
name + '!'
' '.join(['Monty', 'Python']) # joined together with one whitespace as 'glue'
'Monty Python'.split() # splitted into a list

# How can we automatically identify the words of a text that are most informative about the topic and genre of the text? Get the most frequent 50 (mfw = most frequent words)
fdist1 = FreqDist(text1) # frequency distribution
print(fdist1)
fdist1.most_common(50)
fdist1['whale']

Vocab = set(text1)
# get all the words from the vocabulary which are longer than 15 characters as a list
long_words = [w for w in Vocab if len(w) > 15]
# w is an arbitrary shorthand for w
# you could also just say 'word' but it's longer to type it
sorted(long_words)
```

Texts as lists of words

```
# [NLP book, Chap. 3.2] very long words are often hapaxes (i.e., unique) and perhaps it would be better to find frequently occurring long words. This seems promising since it eliminates frequent short words (e.g., the) and infrequent long words (e.g. antiphilosophists). Here are all words from the chat corpus that are longer than seven characters, that occur more than seven times:
sorted(w for w in set(text5) if len(w) > 7 and fdist1[w] > 7)

# [NLP book, Chap. 3.3] A collocation is a sequence of words that occur together unusually often. Thus red wine is a collocation, whereas the wine is not. A characteristic of collocations is that they are resistant to substitution with words that have similar senses; for example, maroon wine sounds definitely odd. To get a handle on collocations, we start off by extracting from a text a list of word pairs, also known as bigrams.
list(bigrams(['more', 'is', 'said', 'than', 'done']))
text4.collocations()
text8.collocations()

fdist1.tabulate()
fdist1.plot()

sorted(w for w in set(text1) if w.endswith('ableness'))
sorted(term for term in set(text4) if 'gnt' in term)
[w.upper() for w in text1] # capitalize words
len(set(word.lower() for word in text1)) # case-folded list of words
```

Nested conditions

```
if len(word) >= 5:
    print('word length is greater than or equal to 5')
for word in ['Call', 'me', 'Ishmael', '.']:
    print(word)

sent1 = ['Call', 'me', 'Ishmael', '.']
for xyzzy in sent1:
    if xyzzy.endswith('l'):
        print(xyzzy)
        print(xyzzy, end=' ') # print not as newline but space-separated
```

Basics of CLTK (Classical Language Toolkit)

`http://cltk.org/` ● CLTK-Installation Page

```
python3 -m venv venv
source venv/bin/activate
pip install cltk
```

`sudo apt-get install python-dev` .... If error (missing Python.h)

```
sudo apt update
sudo apt install git
sudo apt-get install python-setuptools
sudo apt install python-virtualenv
virtualenv -p python3 ~/venv
source ~/venv/bin/activate
pip3 install cltk
```

Install from source: Clone from Github, then:

```
pip install -U -r requirements.txt
python setup.py install
```

In the following code boxes are some example usages. There are many more tutorials on the CLTK Github (accessible through the project web page) such as Lemmatization, Creating a Lexical Dis-perion Plot, etc.

Further CLTK troubleshooting

Setup Tutorial
In den Ordner navigieren, wo CLTK ist ● Jupyter Notebook von dort aus starten, sonst kann es kein CLTK.

J-i u-v replacement

```
# Converting J to I, V to U
from cltk.stem.latin.j_v import JVReplacer

j = JVReplacer()
j.replace('vem jam') #Out[3]: 'uem iam'
```

Declining using Collatinus

```
from cltk.stem.latin.declension import CollatinusDecliner

decliner = CollatinusDecliner()

print(decliner.decline("via"))
decliner.decline("via", flatten=True)
```

Lemmatizer

```
# The CLTK offers a series of lemmatizers that can be combined in a backoff chain, i.e. if one lemmatizer is unable to return a headword for a token, this token can be passed onto another lemmatizer until either a headword is returned or the sequence ends.
from cltk.lemmatize.latin.backoff import BackoffLatinLemmatizer

lemmatizer = BackoffLatinLemmatizer()
tokens = ['Quo', 'usque', 'tandem', 'abutere', ',', 'Catilina', ',', 'patientia', 'nostra', '?']
lemmatizer.lemmatize(tokens)
```

Line Tokenization

```
from cltk.tokenize.line import LineTokenizer

tokenizer = LineTokenizer('latin')
untokenized_text = """49. Miraris verbis nudis me scribere versus?\nHoc brevitas fecit, sensus coniungere binos."""
tokenizer.tokenize(untokenized_text)
```

Stopword Removal

```
from nltk.tokenize.punkt import PunktLanguageVars
from cltk.stop.latin.stops import STOPS_LIST

sentence = 'Quo usque tandem abutere, Catilina, patientia nostra?'
p = PunktLanguageVars()
tokens = p.word_tokenize(sentence.lower())
[w for w in tokens if not w in STOPS_LIST]
# alternative to this Perseus list, a custom stop list can be built, see docs
```

Macronizer

```
# Macronizer: Automatically mark long Latin vowels with a macron.
# Note that the macronizer's accuracy varies depending on which tagger is used.
# macronized text is needed for scansion (see below)

from cltk.prosody.latin.macronizer import Macronizer
macronizer = Macronizer('tag_ngram_123_backoff')

text = 'Quo usque tandem, O Catilina, abutere nostra patientia?'
macronizer.macronize_text(text)
# Out[4]: 'quō usque tandem , ō catilīnā , abūtēre nostrā patientia ?
macronizer.macronize_tags(text) # alternatively
```

CLTK Scansion

```
from cltk.prosody.latin.scanner import Scansion
from cltk.prosody.latin.clausulae_analysis import Clausulae

text = 'quō usque tandem abūtēre, Catilīna, patientiā nostrā. quam diū etiam furor iste tuus nōs ēlūdet.'

s = Scansion()
c = Clausulae()

prosody = s.scan_text(text) #Out[6]: ['-uuu-uuu-u--x', 'uu-uu-uu----x']

c.clausulae_analysis(prosody)
```

Prosody Scanner

```
# A prosody scanner is available for text which already has had its natural lengths marked with macrons. It returns a list of strings of long and short marks for each sentence, with an anceps marking the last syllable of each sentence.
# The algorithm is designed only for Latin prose rhythms. It is detailed in Keeline, T. and Kirby, J "Auceps syllabarum: A Digital Analysis of Latin Prose Rhythm," Journal of Roman Studies, 2019.
from cltk.prosody.latin.scanner import Scansion
scanner = Scansion()
text = 'quō usque tandem abūtēre, Catilīna, patientiā nostrā. quam diū etiam furor iste tuus nōs ēlūdet.'
scanner.scan_text(text)
```

HexameterScanner

```
from cltk.prosody.latin.hexameter_scanner import HexameterScanner
scanner = HexameterScanner()
scanner.scan("impulerit. Tantaene animis caelestibus irae?")
```

PentameterScanner

```
from cltk.prosody.latin.pentameter_scanner import PentameterScanner
scanner = PentameterScanner()
scanner.scan("ex hoc ingrato gaudia amore tibi.")
# for more see http://docs.cltk.org/en/latest/latin.html
# HendecasyllableScanner, Syllabifier, etc.
```

Named Entity Recognition

```
# There is available a simple interface to a list of Latin proper nouns (see repo for how the list was created).
# Ergo: will recognize personal names from the list (thus you can add your own!)
from cltk.tag import ner
from cltk.stem.latin.j_v import JVReplacer

text_str = """ut Venus, ut Sirius, ut Spica, ut aliae quae primae dicuntur esse mangitudinis."""
jv_replacer = JVReplacer()
text_str_iu = jv_replacer.replace(text_str)

ner.tag_ner('latin', input_text=text_str_iu, output_type=list)
```

9 CLTK Pipeline

Corpus Import

```
# available as tutorial Jupyter Notebooks from the CLTK Github
# https://github.com/cltk/tutorials/blob/master/2%20Import%20corpora.ipynb
from cltk.corpus.utils.importer import CorpusImporter
my_latin_downloader = CorpusImporter('latin')
my_latin_downloader.list_corpora # show available corpora

my_latin_downloader.import_corpus('latin_text_latin_library')
my_latin_downloader.import_corpus('latin_models_cltk')

# save text in string variable
# Introduction to Cato's De agricultura
cato_agri_praef = "Est interdum praestare mercaturis rem quaerere, nisi tam
    ↪  periculosum sit, et item foenerari, si tam honestum. Maiores nostri sic
    ↪  habuerunt et ita in legibus posiverunt: furem dupli condemnari, foeneratorem
    ↪  quadrupli. Quanto peiorem civem existimarint foeneratorem quam furem, hinc
    ↪  licet existimare. Et virum bonum quom laudabant, ita laudabant: bonum
    ↪  agricolam bonumque colonum; amplissime laudari existimabatur qui ita
    ↪  laudabatur. Mercatorem autem strenuum studiosumque rei quaerendae existimo,
    ↪  verum, ut supra dixi, periculosum et calamitosum. At ex agricolis et viri
    ↪  fortissimi et milites strenuissimi gignuntur, maximeque pius quaestus
    ↪  stabilissimusque consequitur minimeque invidiosus, minimeque male cogitantes
    ↪  sunt qui in eo studio occupati sunt. Nunc, ut ad rem redeam, quod promisi
    ↪  institutum principium hoc erit."

# See http://docs.cltk.org/en/latest/latin.html#sentence-tokenization
from cltk.tokenize.sentence import TokenizeSentence
tokenizer = TokenizeSentence('latin')
cato_sentence_tokens = tokenizer.tokenize_sentences(cato_agri_praef)
print(len(cato_sentence_tokens)) # 9

for sentence in cato_sentence_tokens:
    print(sentence)
    print()

# Import general-use word tokenizer
from cltk.tokenize.word import nltk_tokenize_words
cato_word_tokens = nltk_tokenize_words(cato_agri_praef)
print(cato_word_tokens)
cato_word_tokens_no_punt = [token for token in cato_word_tokens if token not in
    ↪  ['.', ',', ':', ';']]
# you can remove duplicates by using the set() function on it

# There's a mistake here, though:
# capitalized words ('At', 'Est', 'Nunc') would be counted incorrectly.
# So let's lowercase the input string and try again:

cato_agri_praef_lowered = cato_agri_praef.lower()
cato_word_tokens_lowered = nltk_tokenize_words(cato_agri_praef_lowered)
```

Visualize word frequencies

```
from collections import Counter
# You don't get the unique word forms, but count all tokens

cato_word_counts_counter = Counter(cato_cltk_word_tokens_no_punt)
print(cato_word_counts_counter)

# Lexical diversity is a simple measure of unique words divided by total words.
    ↪  This measures how relatively verbose an author is. Such lexical measures are
    ↪  simple but can be illuminating nevertheless.

# Lexical diversity of our little paragraph
print(len(cato_cltk_word_tokens_no_punt_unique) /
    ↪  len(cato_cltk_word_tokens_no_punt))
# This represents the ratio of unique words to re-reused words
```

Text reuse

```
# Load all of Cicero's De divinatione
import os
# from your own machine (!)
div1_fp =
    ↪  os.path.expanduser('~/cltk_data/latin/text/latin_text_latin_library/cicero/divinatione1.txt')
div2_fp =
    ↪  os.path.expanduser('~/cltk_data/latin/text/latin_text_latin_library/cicero/divinatione2.txt')

with open(div1_fp) as fo:
    div1 = fo.read()

with open(div2_fp) as fo:
    div2 = fo.read()

# We will calculate the Levenstein distance
# See http://docs.cltk.org/en/latest/multilingual.html#text-reuse
from cltk.text_reuse.levenshtein import Levenshtein
lev_dist = Levenshtein()
lev_dist.ratio(div1, div2)

from cltk.text_reuse.comparison import long_substring

# Aen 1.1-6
aen = """arma virumque cano, Troiae qui primus ab oris
Italiam, fato profugus, Laviniaque venit
litora, multum ille et terris iactatus et alto
vi superum saevae memorem Iunonis ob iram;
multa quoque et bello passus, dum conderet urbem,
inferretque deos Latio, genus unde Latinum,
Albanique patres, atque altae moenia Romae."""

# Servius 1.1
serv = """arma multi varie disserunt cur ab armis Vergilius coeperit, omnes tamen
    ↪  inania sentire manifestum est, cum eum constet aliunde sumpsisse principium,
    ↪  sicut in praemissa eius vita monstratum est. per 'arma' autem bellum
    ↪  significat, et est tropus metonymia. nam arma quibus in bello utimur pro
    ↪  bello posuit, sicut toga qua in pace utimur pro pace ponitur, ut Cicero
    ↪  "cedant arma togae" , id est bellum paci. alii ideo 'arma' hoc loco proprie
    ↪  dicta accipiunt, primo quod fuerint victricia, secundo quod divina, tertio
    ↪  quod prope semper armis virum subiungit, ut "arma virumque ferens" et "arma
    ↪  acri facienda viro" . arma virumque figura usitata est ut non eo ordine
    ↪  respondeamus quo proposuimus; nam prius de erroribus Aeneae dicit, post de
    ↪  bello. hac autem figura etiam in prosa utimur. sic Cicero in Verrinis "nam
    ↪  sine ullo sumptu nostro coriis, tunicis frumentoque suppeditato maximos
    ↪  exercitus nostros vestivit, aluit, armavit" . non nulli autem hyperbaton
    ↪  putant, ut sit sensus talis 'arma virumque cano, genus unde Latinum Albanique
    ↪  patres atque altae moenia Romae', mox illa revoces 'Troiae qui primus ab
    ↪  oris'; sic enim causa operis declaratur, cur cogentibus fatis in Latium
    ↪  venerit. et est poeticum principium professivum 'arma virumque cano',
    ↪  invocativum 'Musa mihi causas memora', narrativum 'urbs antiqua fuit'. et
    ↪  professivum quattuor modis sumpsit: a duce 'arma virumque cano', ab itinere
    ↪  'Troiae qui primus ab oris', a bello 'multa quoque et bello passus', a
    ↪  generis successu 'genus unde Latinum'. virum quem non dicit, sed
    ↪  circumstantiis ostendit Aeneam."""

print(long_substring(aen, serv))
```

References

[1] Steven Bird, Ewan Klein, and Edward Loper. *Natural Language Processing with Python*. Sebastopol, 2009.

[2] Matthew L. Jockers. *Macroanalysis. Digital Methods and Literary History*. Chicago, 2013.

[3] Matthew L. Jockers and Ted Underwood. "Text-Mining the Humanities". In: *A New Companion To Digital Humanities*. Ed. by Susan Schreibmann, Ray Siemens, and John Unsworth. Oxford: Wiley Blackwell, 2016, pp. 291–306.

[4] Franco Moretti. *Distant Reading*. London, 2013.

[5] Franco Moretti. *Graphs, Maps, Trees. Abstract Models for a Literary History*. London, 2005.

[6] Malte Rehbein. "Informationsvisualisierung". In: *Digital Humanities. Eine Einführung*. Ed. by Fotis Jannidis, Hubertus Kohle, and Malte Rehbein. Stuttgart: Metzler, 2017, pp. 328–342.

[7] Christof Schöch. "Aufbau von Datensammlungen". In: *Digital Humanities. Eine Einführung*. Ed. by Fotis Jannidis, Hubertus Kohle, and Malte Rehbein. Stuttgart: Metzler, 2017, pp. 223–233.

[8] Christof Schöch. "Quantitative Analyse". In: *Digital Humanities. Eine Einführung*. Ed. by Fotis Jannidis, Hubertus Kohle, and Malte Rehbein. Stuttgart: Metzler, 2017, pp. 279–298.

[9] Steffan Sinclair and Geoffrey Rockwell. "Text Analysis and Visualization: Making Meaning Count". In: *A New Companion To Digital Humanities*. Ed. by Susan Schreibmann, Ray Siemens, and John Unsworth. Oxford: Wiley Blackwell, 2016, pp. 274–290.

[10] Bjarne Stroustrup. *Programming. Principles and Practice Using C++. 2nd Edition*. NY: Addison-Wesley, 2014.

[11] Jun Wu. *The Beauty of Mathematics in Computer Science*. Boca Raton: Chapman and Hall/CRC Press, 2018.